

# Compiling Object Oriented Programs

Mooly Sagiv

Chapter 6.2.9

<http://www.cs.tau.ac.il/~msagiv/courses/wcc10.html>

# Object Oriented Programs

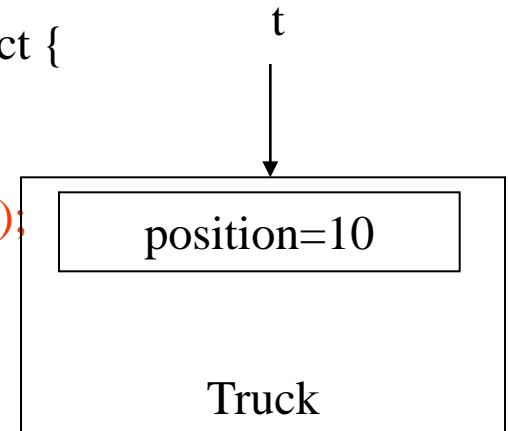
- Objects (usually of type called class)
  - Code
  - Data
- Naturally supports Abstract Data Type implementations
- Information hiding
- Evolution & reusability
- Examples: Simula, Smalltalk, Modula 3, C++, Java, C#, Python

# A Simple Example

```
class Vehicle extends object {  
    int position = 10;  
    void move(int x)  
    {  
        position = position + x ;  
    }  
}
```

```
class Car extends Vehicle {  
    int passengers = 0 ;  
    void await(vehicle v) {  
        if (v.position < position)  
            v.move(position-v.position);  
        else this.move(10);  
    }  
}
```

```
class Truck extends Vehicle {  
    void move(int x)  
    {  
        if (x < 55)  
            position = position+x;  
    }  
}  
class main extends object {  
    void main() {  
        Truck t = new Truck();  
        Car c = new Car();  
        Vehicle v = c;  
        c.move(60);  
        v.move(70);  
        c.await(t);  
    }  
}
```



# A Simple Example

```
class Vehicle extends object {  
    int position = 10;  
    void move(int x)  
    {  
        position = position + x ;  
    }  
}
```

```
class Car extends Vehicle {  
    int passengers = 0 ;  
    void await(vehicle v) {  
        if (v.position < position)  
            v.move(position-v.position);  
        else this.move(10);  
    }  
}
```

```
class Truck extends Vehicle {  
    void move(int x)  
    {  
        if (x < 55)  
            position = position+x;  
    }  
}
```

```
class main extends object {  
    void main() {
```

```
        Truck t = new Truck();
```

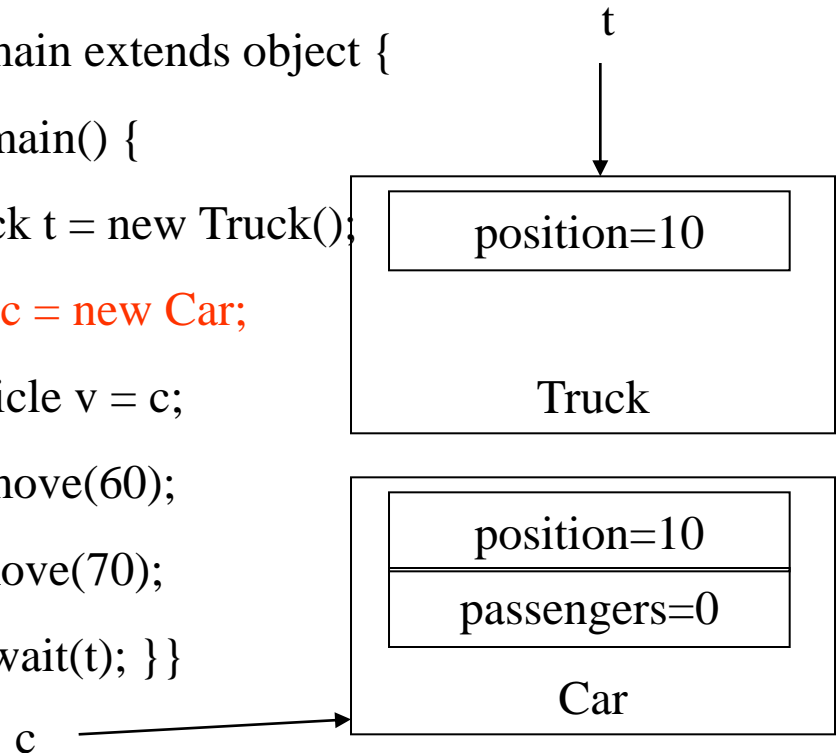
```
        Car c = new Car;
```

```
        Vehicle v = c;
```

```
        c.move(60);
```

```
        v.move(70);
```

```
        c.await(t); } }
```



# A Simple Example

```
class Vehicle extends object {  
    int position = 10;  
    void move(int x)  
    {  
        position = position + x ;  
    }  
}
```

```
class Car extends Vehicle {  
    int passengers = 0 ;  
    void await(vehicle v) {  
        if (v.position < position)  
            v.move(position-v.position);  
        else this.move(10);  
    }  
}
```

```
class Truck extends Vehicle {  
    void move(int x)  
    {  
        if (x < 55)  
            position = position+x;  
    }  
}
```

```
class main extends object {  
    void main() {
```

```
        Truck t = new Truck();
```

```
        Car c = new Car();
```

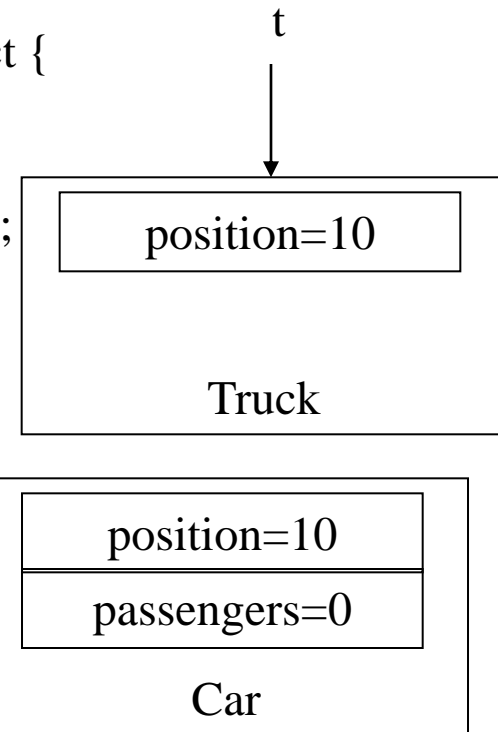
```
        Vehicle v = c;
```

```
        c.move(60);
```

```
        v.move(70);
```

```
        v.c.await(t);}}
```

c



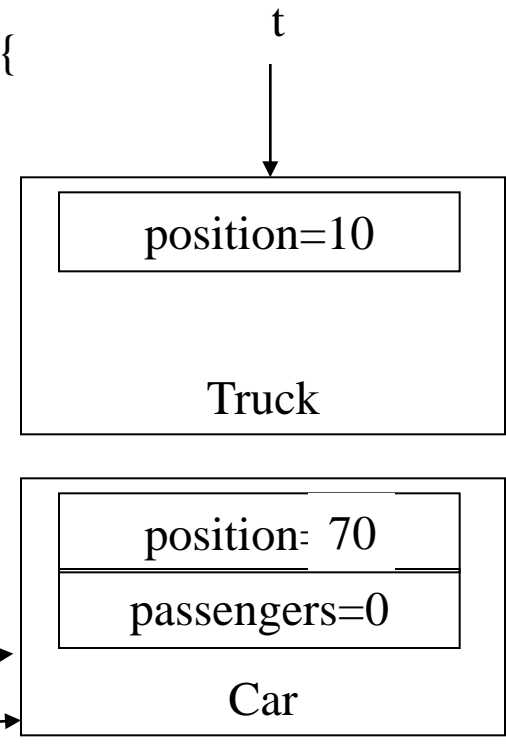
# A Simple Example

```
class Vehicle extends object {  
    int position = 10;  
    void move(int x)  
    {  
        position = position + x ;  
    }  
}
```

```
class Car extends Vehicle {  
    int passengers = 0 ;  
    void await(vehicle v) {  
        if (v.position < position)  
            v.move(position-v.position);  
        else this.move(10);  
    }  
}
```

```
class Truck extends Vehicle {  
    void move(int x)  
    {  
        if (x < 55)  
            position = position+x;  
    }  
}
```

```
class main extends object {  
    void main() {  
        Truck t = new Truck();  
        Car c = new Car();  
        Vehicle v = c;  
        c.move(60);  
        v.move(70);  
        c.await(t);  
    }  
}
```



# A Simple Example

```
class Vehicle extends object {  
    int position = 10;  
    void move(int x)  
    {  
        position = position + x ;  
    }  
}
```

```
class Car extends Vehicle {  
    int passengers = 0 ;  
    void await(vehicle v) {  
        if (v.position < position)  
            v.move(position-v.position);  
        else this.move(10);  
    }  
}
```

```
class Truck extends Vehicle {  
    void move(int x)  
    {  
        if (x < 55)  
            position = position+x;  
    }  
}
```

```
class main extends object {  
    void main() {
```

```
        Truck t = new Truck();
```

```
        Car c = new Car();
```

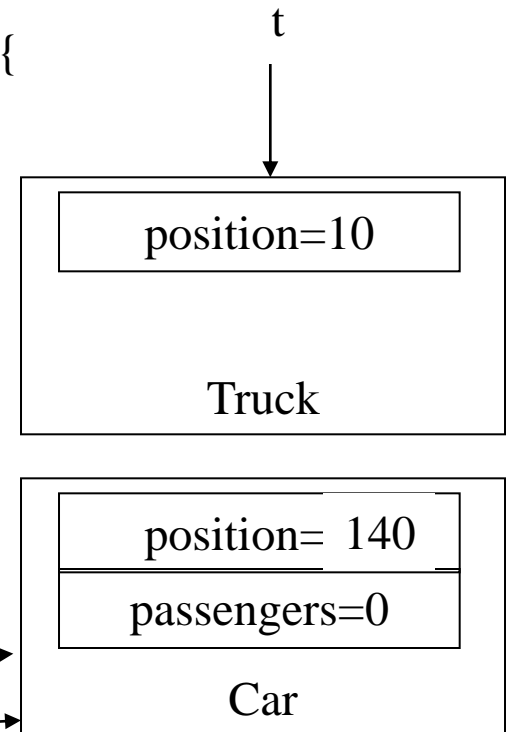
```
        Vehicle v = c;
```

```
        c.move(60);
```

```
        v.move(70);
```

```
        c.await(t);}}
```

v  
c



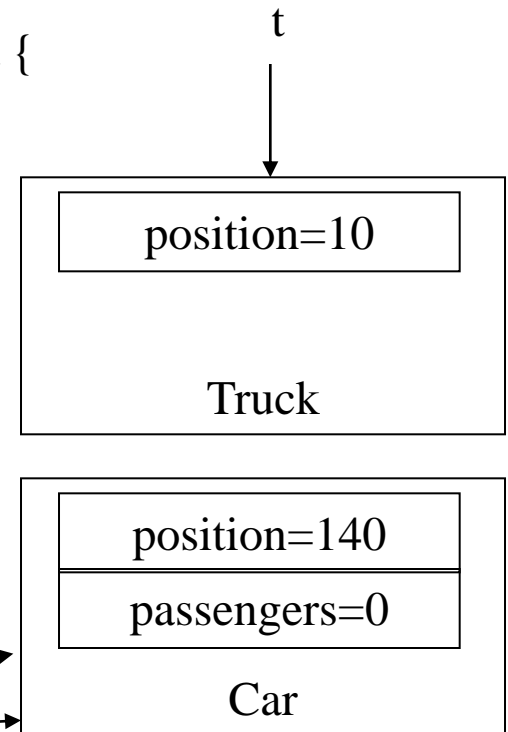
# A Simple Example

```
class Vehicle extends object {  
    int position = 10;  
    void move(int x)  
    {  
        position = position + x ;  
    }  
}
```

```
class Car extends Vehicle {  
    int passengers = 0 ;  
    void await(vehicle v) {  
        if (v.position < position)  
            v.move(position-v.position);  
        else this.move(10);  
    }  
}
```

```
class Truck extends Vehicle {  
    void move(int x)  
    {  
        if (x < 55)  
            position = position+x;  
    }  
}
```

```
class main extends object {  
    void main() {  
        Truck t = new Truck();  
        Car c = new Car();  
        Vehicle v = c;  
        c.move(60);  
        v.move(70);  
        c.await(t);  
    }  
}
```





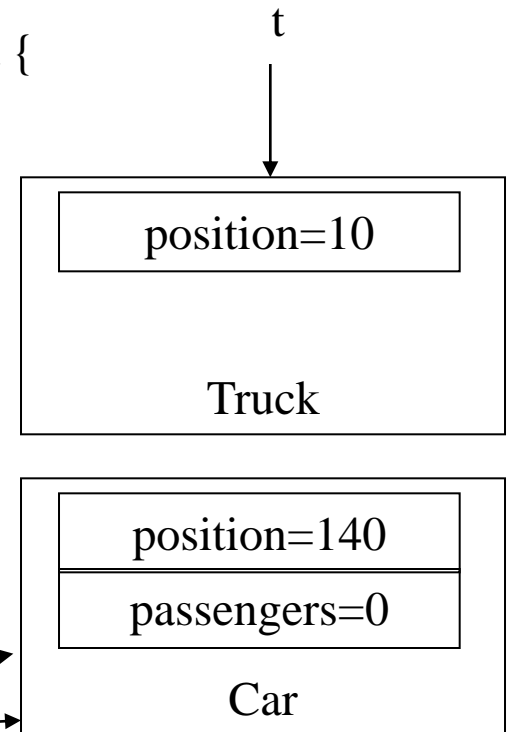
# A Simple Example

```
class Vehicle extends object {  
    int position = 10;  
    void move(int x)  
    {  
        position = position + x ;  
    }  
}
```

```
class Car extends Vehicle {  
    int passengers = 0 ;  
    void await(vehicle v) {  
        if (v.position < position)  
            v.move(position-v.position);  
        else this.move(10);  
    }  
}
```

```
class Truck extends Vehicle {  
    void move(int x)  
    {  
        if (x < 55)  
            position = position+x;  
    }  
}
```

```
class main extends object {  
    void main() {  
        Truck t = new Truck();  
        Car c = new Car();  
        Vehicle v = c;  
        c.move(60);  
        v.move(70);  
        c.await(t);  
    }  
}
```



# Translation into C (Vehicle)

```
class Vehicle extends object {  
    int position = 10;  
    void move(int x)  
    {  
        position = position + x ;  
    }  
}
```

```
struct Vehicle {  
    int position ;  
}  
void New_V(struct Vehicle *this)  
{  
    this->position = 10;  
}  
void move_V(struct Vehicle *this, int x)  
{  
    this->position=this->position + x;  
}
```

# Translation into C(Truck)

```
class Truck extends Vehicle {  
  void move(int x)  
  {  
    if (x < 55)  
      position = position+x;  
  }  
}
```

```
struct Truck {  
  int position ;  
  }  
void New_T(struct Truck *this)  
{  
  this->position = 10;  
}  
void move_T(struct Truck *this, int x)  
{  
  if (x <55)  
    this->position=this->position + x;  
}
```

# Naïve Translation into C(Car)

```
class Car extends Vehicle {  
    int passengers = 0 ;  
    void await(vehicle v) {  
        if (v.position < position)  
            v.move(position-v.position);  
        else this.move(10);  
    }  
}
```

```
struct Car {  
    int position ;  
    int passengers;    }  
void New_C(struct Car *this)  
{ this->position = 10;  
  this ->passengers = 0;  }  
void await_C(struct Car *this, struct Vehicle *v)  
{ if (v->position < this ->position )  
    move_V(this ->position - v->position )  
  else Move_C(this, 10) ;}
```

# Naïve Translation into C(Main)

```
class main extends object {  
  void main(){  
    Truck t = new Truck();  
    Car c = new Car();  
    Vehicle v = c;  
    c.move(60);  
    v.move(70);  
    c.await(t);} }  
  
void main_M()  
{  
  struct Truck *t = malloc(1, sizeof(struct Truck));  
  struct Car *c= malloc(1, sizeof(struct Car));  
  struct Vehicle *v = (struct Vehicle*) c;  
  move_V((struct Vehicle*) c, 60);  
  move_V(v, 70);  
  await_C(c,(struct Vehicle *) t);  
}
```

# Compiling Simple Classes

- Fields are handled as records
- Methods have unique names

	Runtime object	Compile-Time Table				
class A {	<table border="1"><tr><td>a1</td></tr><tr><td>a2</td></tr></table>	a1	a2	<table border="1"><tr><td>m1_A</td></tr><tr><td>m2_A</td></tr></table>	m1_A	m2_A
a1						
a2						
m1_A						
m2_A						
field a1;						
field a2;						
method m1() {...}						
method m2(int i) {...}	void m2_A(class_A *this, int i)					
}	{	Body of m2 with any object field x as this →x				
a.m2(5)		m2_A(&a, 5)				

# Features of OO languages

- Inheritance
- Method overriding
- Polymorphism
- Dynamic binding

# Handling Single Inheritance

- Simple type extension
- Type checking module checks consistency
- Use prefixing to assign fields in a consistent way

```
class A {  
    field a1;  
    field a2;  
    method m1() {...}  
    method m2() {...}  
}
```

Runtime object

a1
a2

Compile-Time Table

m1_A
m2_A

```
class B extends A {  
    field a3;  
    method m3() {...}  
}
```

Runtime object

a1
a2
a3

Compile-Time Table

m1_A
m2_A
m3_B



# Method Overriding

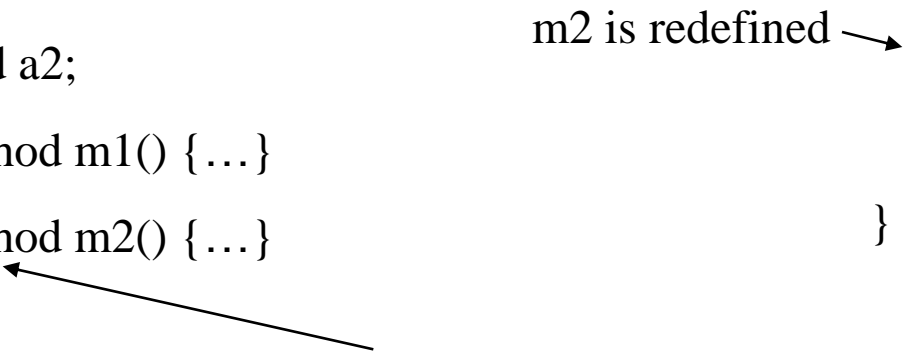
- Redefines functionality

```
class A {  
    field a1;  
    field a2;  
    method m1() {...}  
    method m2() {...}  
}
```

m2 is declared and defined

```
class B extends A {  
    field a3;  
    method m2() {...}  
    method m3() {...}  
}
```

m2 is redefined



# Method Overriding

- Redefines functionality
- Affects semantic analysis

```
class A {  
    field a1;  
    field a2;  
    method m1() {...}  
    method m2() {...}  
}
```

```
class B extends A {  
    field a3;  
    method m2() {...}  
    method m3() {...}  
}
```

Runtime object

a1
a2

Compile-Time Table

m1_A_A
m2_A_A

Runtime object

a1
a2
a3

Compile-Time Table

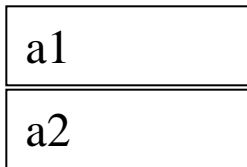
m1_A_A
m2_A_B
m3_B_B

# Method Overriding

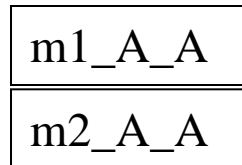
```
class A {  
    field a1;  
    field a2;  
    method m1() {...}  
    method m2() {...}  
}
```

```
class B extends A {  
    field a3;  
    method m2() {...}  
    method m3() {...}  
}
```

Runtime object



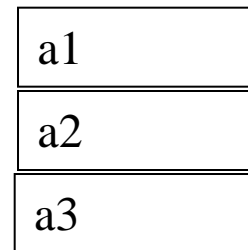
Compile-Time Table



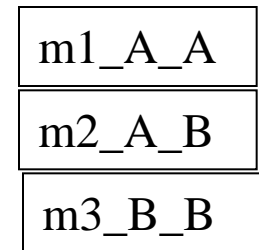
a.m2 () // class(a)=A

↓  
m2\_A\_A (&a)

Runtime object



Compile-Time Table



a.m2 () // class(a)=B

↓  
m2\_A\_B (&a)

# Method Overriding (C)

```
struct class_A {  
    field a1;  
    field a2;  
}  
  
void m1_A_A(class_A *this) {...}  
void m2_A_A(class_A *this, int x) {...}
```

```
struct class_B {  
    field a1;  
    field a2;  
    field a3;  
}  
  
void m2_A_B(class_B *this, int x) {...}  
void m3_B_B(class B *this) {...}
```

Runtime object

a1
a2

Compile-Time Table

m1_A_A
m2_A_A

Runtime object

a1
a2
a3

Compile-Time Table

m1_A_A
m2_A_B
m3_B_B

a.m2 (5) // class(a)=A

⇓  
m2\_A\_A (&a, 5)

a.m2 (5) // class(a)=B

⇓  
m2\_A\_B (&a, 5)

# Abstract Methods

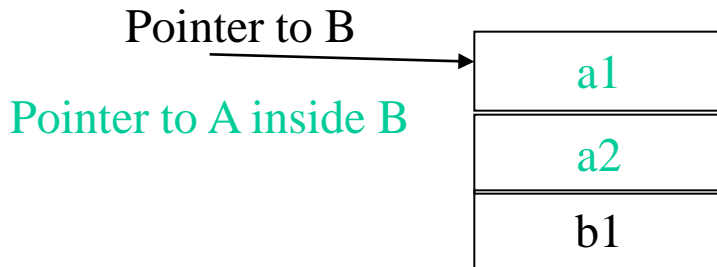
- Declared separately
  - Defined in child classes
- Java abstract classes
- Handled similarly
- Textbook uses “Virtual” for abstract

# Handling Polymorphism

- When a class **B** extends a class **A**
  - variable of type pointer to A may actually refer to object of type B
- Upcasting from a subclass to a superclass
- Prefixing guarantees validity

```
class B *b = ...;
```

```
class A *a = b ;      =>  class A *a=convert_ptr_to_B_to_ptr_A(b) ;
```



# Dynamic Binding

- An object **o** of class **A** can refer to a class **B**
- What does ‘**o.m()**’ mean?
  - Static binding
  - Dynamic binding
- Depends on the programming language rules
- How to implement dynamic binding?
- The invoked function is not known at compile time
- Need to operate on data of the **B** and **A** in consistent way

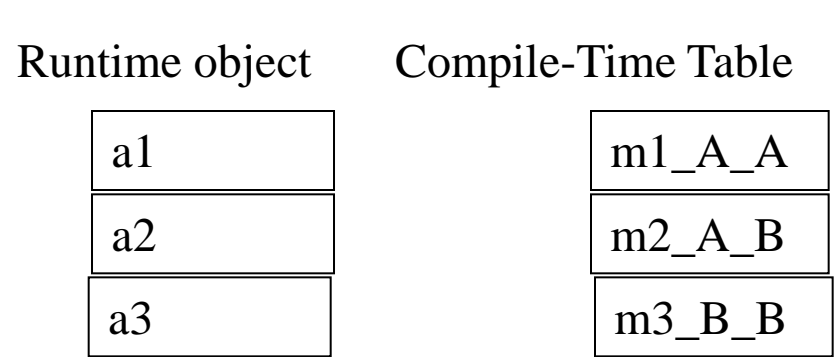
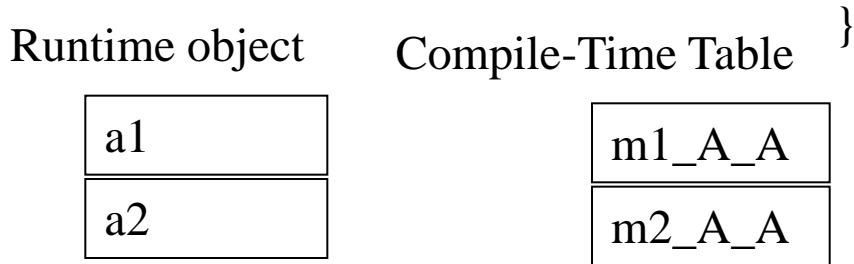
# Conceptual Implementation of Dynamic Binding

```
struct class_A {
    field a1;
    field a2;
}

void m1_A_A(class_A *this) {...}
void m2_A_A(class_A *this, int x) {...}
```

```
struct class_B {
    field a1;
    field a2;
    field a3;
}

void m2_A_B(class_B *this, int x) {...}
void m3_B_B(class B *this) {...}
```



```
switch(dynamic_type(p) {
    case Dynamic_class_A: m2_A_A(p, 3);
    case Dynamic_class_B: m2_A_B(convert_ptr_to_A_to_ptr_B(p), 3);
}
```



# More efficient implementation

- Apply pointer conversion in subclasses

```
void m2_A_B(class A *this_A, int x) {  
    Class_B *this = convert_ptr_to_A_ptr_to_A_B(this_A);  
    ...  
}
```

- Use dispatch table to invoke functions
- Similar to table implementation of case

```

struct class_A {
    field a1;
    field a2;
}

void m1_A_A(class_A *this) { ... }
void m2_A_A(class_A *this, int x)
{ ... }

```

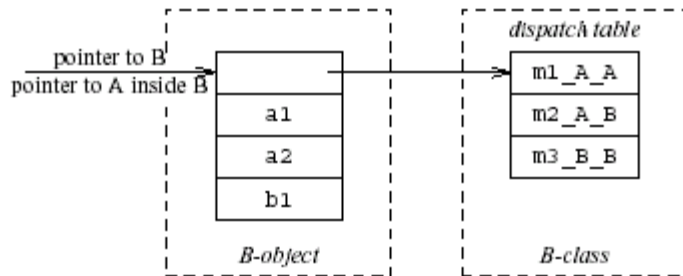
```

struct class_B {
    field a1;
    field a2;
    field a3;
}

void m2_A_B(class_A *this_A, int x) {
    Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(this_A);
    ... }

void m3_B_B(class A *this_A) { ... }
}

```



p.m2(3);

p→dispatch\_table→m2\_A(p, 3);

```

struct class_A {
    field a1;
    field a2;
}

void m1_A_A(class_A *this) { ... }
void m2_A_A(class_A *this, int x)
{ ... }

```

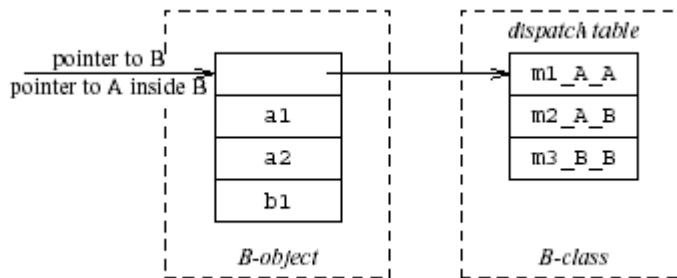
```

struct class_B {
    field a1;
    field a2;
    field a3;
}

void m2_A_B(class_A *this_A, int x) {
    Class_B *this =
    convert_ptr_to_A_to_ptr_to_B(this_A);
    ... }

void m3_B_B(class A *this_A) { ... }
}

```



p.m2(3); // p is a pointer to B

m2\_A\_B(convert\_ptr\_to\_B\_to\_ptr\_to\_A(p), 3);

# Multiple Inheritance

```
class C {  
    field c1;  
    field c2;  
    method m1();  
    method m2();  
};  
  
class D {  
    field d1;  
    method m3();  
    method m4();  
};  
  
class E extends C, D {  
    field e1;  
    method m2();  
    method m4();  
    method m5();  
};
```

# Multiple Inheritance

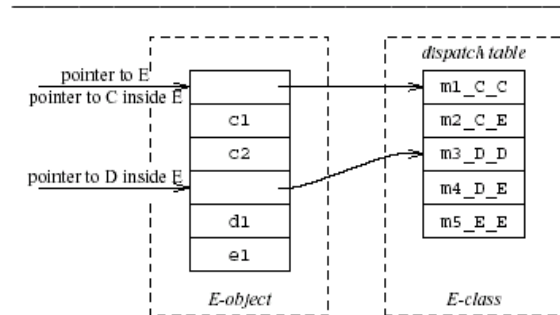
- Allows unifying behaviors
- But raises semantic difficulties
  - Ambiguity of classes
  - Repeated inheritance
- Hard to implement
  - Semantic analysis
  - Code generation
    - Prefixing no longer work
    - Need to generate code for downcasts
- Hard to use

# A simple implementation

- Merge dispatch tables of superclasses
- Generate code for upcasts and downcasts

# A simple implementation

```
class C {  
    field c1;  
    field c2;  
    method m1();  
    method m2();  
};  
  
class D {  
    field d1;  
    method m3();  
    method m4();  
};  
  
class E extends C, D {  
    field e1;  
    method m2();  
    method m4();  
    method m5();  
};
```

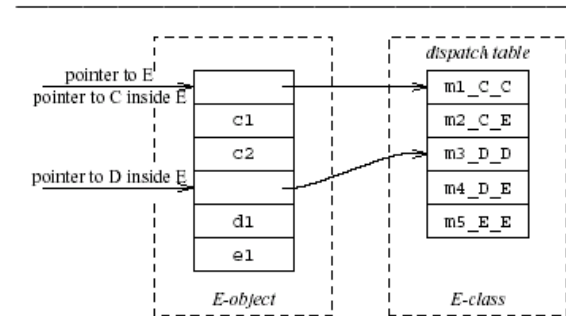


# A simple implementation (downcasting)

```
class C {
    field c1;
    field c2;
    method m1 ();
    method m2 ();
};

class D {
    field d1;
    method m3 ();
    method m4 ();
};

class E extends C, D {
    field e1;
    method m2 ();
    method m4 ();
    method m5 ();
};
```



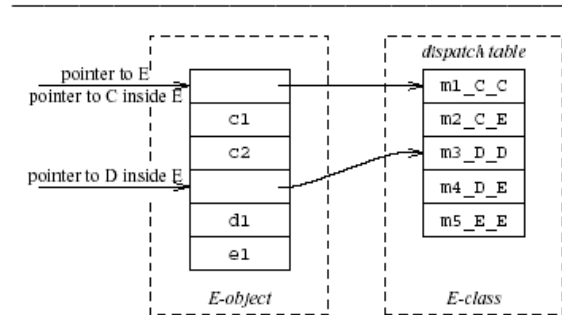
`convert_ptr_to_E_to_ptr_to_C(e) = e;`

`convert_ptr_to_E_to_ptr_to_D(e) = e + sizeof(C);`



# A simple implementation (upcasting)

```
class C {  
    field c1;  
    field c2;  
    method m1 ();  
    method m2 ();  
};  
  
class D {  
    field d1;  
    method m3 ();  
    method m4 ();  
};  
  
class E extends C, D {  
    field e1;  
    method m2 ();  
    method m4 ();  
    method m5 ();  
};
```



`convert_ptr_to_C_to_ptr_to_E(c) = c;`

`convert_ptr_to_D_to_ptr_to_E(d) = d - sizeof(C);`

# Dependent Multiple Inheritance

```
class A {
    field a1;
    field a2;
    method m1();
    method m3();
};

class C extends A {
    field c1;
    field c2;
    method m1();
    method m2();
};

class D extends A {
    field d1;
    method m3();
    method m4();
};

class E extends C, D {
    field e1;
    method m2();
    method m4();
    method m5();
};
```

# Dependent Inheritance

- The simple solution does not work
- The positions of nested fields do not agree

# Implementation

- Use an index table to access fields
- Access offsets indirectly
- Some compilers avoid index table and use register allocation techniques to globally assign offsets

```

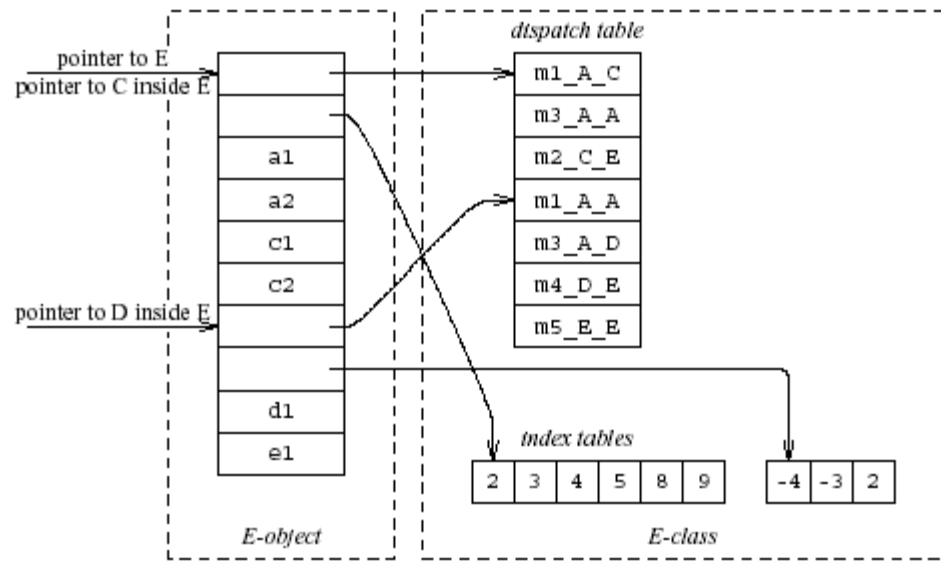
class A {
    field a1;
    field a2;
    method m1();
    method m3();
};

class C extends A {
    field c1;
    field c2;
    method m1();
    method m2();
};

class D extends A {
    field d1;
    method m3();
    method m4();
};

class E extends C, D {
    field e1;
    method m2();
    method m4();
    method m5();
};

```



# Class Descriptors

- Runtime information associated with instances
- Dispatch tables
  - Invoked methods
- Index tables
- Shared between instances of the same class

# Interface Types

- Java supports limited form of multiple inheritance
- Interface consists of several methods but no fields
- A class can implement multiple interfaces

```
public interface Comparable {  
    public int compare(Comparable o);  
}
```
- Simpler to implement/understand/use
- A separate dispatch table per interface specification which refers to the implemented method

# Dynamic Class Loading

- Supported by some OO languages (Java)
- At compile time
  - the actual class of a given object at a given program point may not be known
- Some addresses have to be resolved at runtime
- Compiling  $c.f()$  when  $f$  is dynamic:
  - Fetch the class descriptor  $d$  at offset  $0$  from  $c$
  - Fetch  $p$  the address of the method-instance  $f$  from (constant)  $f$  offset at  $d$
  - Jump to the routine at address  $p$  (saving return address)



# Other OO Features

- Information hiding
  - private/public/protected fields
  - Semantic analysis (context handling)
- Testing class membership

# Optimizing OO languages

- Hide additional costs
- Replace dynamic by static binding when possible
- Eliminate runtime checks
- Eliminate dead fields
- Simultaneously generate code for multiple classes
- Code space is an issue

# Summary

- OO features complicates compilation
  - Semantic analysis
  - Code generation
  - Runtime
  - Memory management
- Understanding compilation of OO can be useful for programmers